Solutions of versions of the ‘Countdown numbers game’.

Elliot Fenner

Computer Science and Philosophy

2000/2001

(20 Credits)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

Signature_____

Summary

In this project I investigated the numbers game from the television programme “Countdown”. In this game, a contestant is given six numbers from which they have to construct an arithmetic expression (using each number at most once) using the four standard operators to obtain a given target number. I considered different algorithms for solving the problem, both the Classical game as in the television programme, and a generalised version. The aim of the project was to produce a working program to solve both versions of the game. I produced a user-friendly version with a graphical interface (GUI) for the Classical game, and a command line interface which can be used with both versions of the game. I also wrote a driver for the command line version to obtain some statistical information about the game using random data. I also looked at the run time complexity and compared this with actual run times for different sizes of the problem. I then analysed my results to see if there were any interesting and significant trends.

Acknowledgements

I would like to thank my project supervisor Haiko Muller for his guidance, advice and support during the project.

I would also like to thank all my housemates for their help with testing the program.

Special thanks go to my parents for their encouragement and love all these years.

Contents

Summary ii

Acknowledgements iii

1. Introduction 1
1.1 Reasons for choosing project 1
1.2 Description of the Problem 1

2. Background and Specification of the Problem 2
2.1 Background Information 2
2.2 Classical Problem 3
2.3 Generalised Problem 4
2.4 Complexity 4
2.5 Solution 5
2.6 Problem Scope 5

3. Project Objectives 7
3.1 Aim 7
3.2 Objectives 7
3.3 Minimum Requirements 7

4. Algorithm Design 8
4.1 Possible Techniques 8
4.1.1. Subset Sum Problem 8
4.1.2. Countdown Problem 10
4.2 Choice of Algorithm 11
4.3 Optimisations 12
4.4 Other Issues 14

5. Program Design 18
5.1 General Issues 18
5.2 Command Line version 20
5.3 GUI version 21

6. Complexity Issues 22
6.1 Complexity of the Problem 22
6.2 Complexity of the Algorithms 22
6.2.1. Un-optimised version 22
6.2.2. Optimised version 23

7. Testing **27**
7.1 Analysis of sample data 27
7.2 Conclusions from sample data 29

8. Results and Conclusion **32**
8.1 Does the project meet the objectives? 32
8.2 Further Work 33

References **34**

Appendices
Appendix A - Lessons Learnt
Appendix B – Tree Diagrams
Appendix C - Example runs of Countdown
Appendix D – Screen Shots
Appendix E – Graphs and Tables of Results
Appendix F – User manual
Appendix G – Program Listings

1. Introduction

1.1 Reasons for choosing the project

I chose this project as I wanted to do a project which gave me scope to design and implement algorithms, especially recursive ones. From my course 'Functional Programming (CO21)' last year, I saw that recursion is a very elegant and powerful technique and I wanted to tackle a problem making use of it.

I had never constructed a GUI before, so I wanted to choose a project which gave me practice in designing and creating a simple GUI. I also wanted to gain some experience in using JAVA as I had learnt it in my 'Object Oriented Programming (SO21)' course last year, but have never really had to construct a large program using it. I thought that JAVA would also be a good platform to use for creating the GUI.

I enjoy watching the Countdown programme on television, trying to solve the numbers game problems faster than the contestants. I wanted to look at the efficiency of any algorithm I produced and see whether there was scope for improving it.

I would also be able to use the knowledge learnt in the "Algorithms and Complexity (CO33)" course to assist me in algorithm analysis and complexity.

1.2 Description of Problem

The problem is to design at least one algorithm to solve the Countdown problem: Given 6 positive integers and a target integer, the question is whether it is possible to construct an arithmetic expression, using just the standard four operators, which evaluates to the target and uses each of the 6 numbers at most once.

2. Background and specification of the problem

2.1 Background Information

There are several issues which I had to read up on to get a better understanding of the problem and its complexity and possible methods for solving it:

- The complexity classes P, NP and NP-C (where NP-C is the class of NP-complete problems)
- The Subset sum problem

The complexity classes P, NP and NP-complete

The following definitions are essentially based on those in Baase. The complexity classes are all classes of decision problems. A decision problem is a question that has two possible answers, *yes* and *no*. The question is about some input. A *problem instance* is the combination of the problem and a specific input. Usually the statement of a decision problem has two parts:

1. The *instance description* part defines the information expected in the input.
2. The *question* part states the actual yes-or-no question; the question contains variables defined in the instance description.

A decision problem's output is either *yes* or *no* according to the correct answer to the question, as applied to a given input. Thus a decision problem can be thought of abstractly as a mapping from inputs to the set {*yes*, *no*}. An algorithm is said to be *polynomially bounded* if its worst case time complexity is bounded by a polynomial function of the input size. A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it. P is the class of problems that are polynomially bounded. NP is more complicated to describe.

Many decision problems are phrased as existence questions: Does there exist a k-colouring of the graph G? For a given input, a "solution" is an object that satisfies the criteria in the problem and hence justifies a *yes* answer. NP is the class of decision problems for which there is a polynomially bounded *nondeterministic* algorithm.

A *nondeterministic algorithm* has two phases and an output step.

1. The nondeterministic "guessing" phase. Some completely arbitrary string of characters, *s*, is written ... Each time the algorithm is run, the string written may differ. (...it may be thought of as a guess at a solution for the problem, so this phase may be called the guessing phase, but *s* could just as well be gibberish).

2. The deterministic "verifying" phase. A deterministic subroutine begins execution. In addition to the decision problem's input, the subroutine may use *s*, or it may ignore *s*. Eventually it returns a value *true* or *false*, or it may go in to an infinite loop and never halt (the verifying phase can be thought as checking *s* to see if it is a solution for the decision problem's input, i.e., if it justifies a *yes* answer for the decision problem's input.)

3. The output step. If the verifying phase returned true, the algorithm outputs *yes*. Otherwise, there is no output.

NP-complete is the term used to describe decision problems that are the hardest ones in NP in the sense that, if there were a polynomially bounded algorithm for an NP-complete problem, then there would be a polynomially bounded algorithm for each problem in NP.

The Subset sum problem

The subset sum problem is defined as follows.

Instance: A positive integer T and n (not necessary distinct) positive integers whose sizes are s_1, s_2, …s_n.

Question: Is there a subset I of the set {1, 2,…, n} such that $\sum_{i \in I} s_i = T$?

It has been proved that the subset sum problem is NP-complete. This can be shown by transforming 3 Dimensional Matching, which is known to be NP-complete, to the subset sum problem (Garey and Johnson).

2.2 Classical Problem

We will refer to the version of the numbers game used in the television programme as the Classical *problem*. The Countdown numbers game is a kind of extension of the subset sum problem. In the numbers game, you are given a positive target T and 6 other (not necessarily distinct) positive integers $s_{1,\ldots,} s_6$. The problem is to find an arithmetic expression in which you may use the four standard operators {+, -, x, ÷} and the six numbers, s_1, s_2, …, s_6, whose value is equal to the target T. The six numbers do not all have to be used, but no number may be used more than once (unless the number occurs more than once in s_1, s_2, …, s_6).

The selection of the six numbers takes place as follows. There are 24 cards. Four contain the *big* numbers 25,50,75,100. The remaining 20 cards contain the *small* numbers 1,2,3,4,5,6,7,8,9,10. Each of

the small numbers appears twice whereas the big numbers appear only once each. The player has to choose 6 cards in total from either the small or big sets. Although they can decide whether to choose a card from either the small or big set, they do not know what the number is (so the possible choices are, 6 small, 5 small 1 big, 4 small 2 big, 3 small 3 big, 2 small 4 big). A target integer, between 101 and 999 inclusive, is then chosen at random. Each player has to try to find an expression giving the desired target.
If one contestant reaches the target but the other doesn't then that contestant wins. If both contestants reach the target, then it is a draw. If neither contestant can reach the target exactly, then the contestant whose expression is closest to the target, either above or below, wins. If they are both the same distance away from the target it is also a draw.

For example, if the target is 538 and the given numbers are 2, 3, 7, 9, 10 and 75.
One possible solution could be (((75+2) * 7) + 9) - 10.
However, if the target is 951 and the given numbers are 1, 3, 4, 5, 7 and 9, then there is no solution and the nearest values attainable are 952 and 950.

There are some additional rules for the Countdown numbers game:

Only integers are allowed to be used, even as an intermediate result.
E.g. if the target is 850 and the numbers are, 100, 75, 50, 10, 3, 6.
It would not be possible to use the expression (((75/10) + (3/6)) * 100) + 50 to obtain 850.

Similarly, negative numbers are not allowed to be used as intermediate results.

2.3 Generalised Problem

As well as the classical problem I will look at a generalised version of the game, using n integers, s_1, s_2, …, s_n, as opposed to just the six which are used in the Classical game. I will also allow a wider range of numbers to be chosen.

The *subset sum problem* is a restriction of the generalised problem, to the case where the only operator that can be used is '+'; so the generalised problem is an extension of the subset sum problem. I will also

allow versions where the permitted operators are an arbitrary subset of the four standard operators. From now on, we will refer to the generalised problem as the *Countdown problem.*

2.4 Complexity

At first sight it appears that the Countdown problem will be NP-complete as it is an extension of the subset sum problem since by adding another 3 operators, the problem only gets larger and more complicated. However, it could be argued that although there now exist many more possibilities which have to be considered, there are many more ways by which it may be possible to reach the target.

I discovered that that the subset product problem (where only multiplication is allowed) is also NP-complete (Garey & Johnson, although no proof is given in the text, a proof is included later in Section 6).

There does not seem to be any easy way to prove that the Countdown problem is NP-complete when all four operators are allowed. However, it would be very surprising if it turned out that it is not NP-complete.

2.5 Solution

On the assumption that the problem is NP-complete, the solution will have to be obtained essentially by some kind of exhaustive search (i.e. every possibility has to be considered in some way).

Three techniques, which can be used to solve the subset sum problem, are
a) Recursion
b) Dynamic programming
c) Constraint Programming

I will describe these techniques in more detail later in Section 4 – Algorithm Design.

2.6 Problem Scope

I will be carrying out the following in order to solve and analyse the Countdown problem.

- Producing a command line interface for the generalised version. This will allow users to decide how many numbers they wish to use for the problem, as well as choosing whether they wish to select the numbers and target themselves, or generate them randomly.

- Producing a GUI for the Classical problem.

- Investigating the Classical and various cases of the generalised problem by experimentation with randomly generated data; for example, to find out how many solutions there are for different combinations of large and small numbers. I will also try and find the probability of reaching the target when it is reachable. Different cases will be considered, such as:
  if all the numbers chosen are small (e.g., 6 small numbers);
  if all the large numbers are chosen (e.g. 4 large numbers and 2 small numbers);
  I will also investigate for randomly generated number sets, how many different targets can be reached.

- Optimisation; we will see later that using some methods the same sub-expression may be computed several times. Therefore, if necessary, I will try to modify the algorithm to avoid repeated computations where possible.

- Analysing the complexity of the algorithm, or at least get an upper bound on the complexity, for both the simple and improved versions of the algorithm.

- Carrying out timing tests and comparing the results with the theoretical complexity.

3. Project Objectives

3.1 Aim

The aim of this project is to provide a solution to both Classical and generalised versions of the Countdown numbers game (from the television programme), and to provide both a command line and GUI version of the program. Investigating the behaviour of the running time of the program, to investigate the existence and number of solutions, using random problem instances.

3.2 Objectives

To write and implement one or more algorithms which attempt to find an exact solution of the numbers game where one exists, or otherwise a best approximate solution (using a slightly 'smaller' version of the problem if runtime considerations make the solution of the full version infeasible).

To analyse the complexity of the algorithms constructed, and if possible to analyse the complexity of some versions – generalisations and/or restrictions – of the Countdown numbers game.

To make a simple user-friendly graphical front end for the program.

To run the program on various types of random data sets to find out the empirical probability that there exists an exact solution, and to see if there is a difference in the likelihood of reaching that target depending on how many numbers are used.

If time permits, to investigate possible heuristic methods for instances of the problem which are too long for exhaustive search methods to be feasible.

3.3 Minimum Requirements

To write and implement one or more algorithms which attempt to find an exact solution of the numbers game where one exists, or otherwise a best approximate solution (using a slightly 'smaller' version of the problem if runtime considerations make the solution of the full version infeasible).

To analyse the complexity of the algorithms constructed, and if possible to analyse the complexity of some versions – generalisations and/or restrictions – of the numbers game from the television programme 'countdown'.

To make a simple user-friendly front end for the program.

4. Algorithm Design

4.1 Possible Techniques

4.1.1. Subset Sum Problem

As mentioned in Chapter 2, there are at least 3 different possible approaches that one could use to solve the subset sum problem. The first two are standard algorithms which people use to solve problems of this sort. As I am currently doing a module in Constraint Satisfaction and Constraint Programming, I also tried to think of a way to solve the problem using the skills that I had learnt on that course.

I use the term number set or set of numbers to denote a *multiset*, i.e. where repeats are allowed. Similarly a subset of a multiset may also be a multiset. I will denote the number set containing s_1, s_2, .., s_n by $[s_1, s_2, .., s_n]$.

Recursion

Suppose for the subset sum problem with target T and number set $[s_1, s_2, .., s_n]$, a solution Q exists, i.e. the sum of the members of Q is T.
Consider the last number, s_n. Either s_n is one of the numbers in Q or not. If s_n is in Q, then this means that the remaining numbers add up to T-s_n. So the subset sum problem with target T-s_n and number set $[s_1,.., s_{n-1}]$must have a solution. If on the other hand s_n is not in Q, then the subset sum problem with target T and number set $[s_1, .., s_{n-1}]$must have a solution.

We use this as a basis for a recursive algorithm to solve the subset sum problem. If the subset sum problem has a solution then the function returns the value q, where q is the size of some subset Q, which is a solution to the problem. If there is no solution we return the value –1. It also places the elements of Q in the array term[1…q].

```
int sum(int[] s, int n, int target, int[] term)
        int result;
        if (target = 0) return 0;  //solution where Q is empty set
        if (target < 0) return -1;  //no solution
        if (n = 0) return -1;  //no solutions
```

```
result = sum(s, n-1, target, term);   //not using s_n
if (result >= 0)  return result;  //not using s_n

result = sum(s, n-1, target-s[n], term);   // using s_n
if (result < 0) return –1;
term[result+1] = s[n];  //solution using s_n
return result+1;
```

It first tries to find a solution without using s_n, then tries to find a solution using s_n.
If we wanted to find all solutions, we would have to change the program so that both recursive calls are always carried out even if a solution was found on the first recursive call (i.e. not using s_n).

Dynamic Programming

In recursive programs, very often the same recursive call will be executed many times. The dynamic programming approach overcomes this by storing the result of recursive calls on sub-problems rather than re-computing them. For the subset sum problem we record the set R_j of all the targets which are reachable as a sum of the subset $[s_1, \ldots, s_j]$. We then find the set of targets R_{j+1} reachable using $[s_1, \ldots, s_{J+1}]$. This is simply all the elements of R_j together with all the integers $r + s_{j+1}$ for all $r \in R_j$.
We start off with R_0 equal to $\{0\}$. Then the subset sum problem has a solution if the target is in R_n.
An economical way of representing the set is to use an array of bits, where the length of the array is at least the largest possible target, i.e. $s_1 + s_2 + \ldots + s_n$ and the *ith* bit in the bit array is set to 1 if *i* is in the set, or zero if *i* is not in the set.

Constraint Programming

It is also possible to think of the subset sum problem as a constraint problem. In the subset sum problem a particular number is either used in the sum or not. We can represent a set by using an array X of indicator variables, where X[i] = 1 if A[i] is included in the sum, otherwise X[i] = 0.

```
//All pseudo-code is based on Solver
//The variable m is the manager for handling input, output and memory allocation of
//constrained variables.
```

```
IlcIntArray A (m, n);          // A is an array of n integers in the set
IlcIntVarArray X (m, n, 0, 1); // X is an array of n integers having values 0 or 1
IlcIntVar Sum;                 // The sum of the included integers
IlcInt Target;                 // The value of the target

Sum = IlcScalProd (A, X);      // Inner product i.e.
                               // A[0] * X[0] + A[1] * X[1] + …+ A[n-1] * X[n-1]

m.add(Sum == Target);   // add the constraint that Sum = Target
```

* * *

In all these three approaches, it would be easy to extend the above methods for dealing with expressions with both '+' and '-'. For example, in the constraint programming approach you would simply have to change the range that the values of X[i] take to be –1, 0 or 1, viz.

IlcIntVarArray X(m,n,-1,1).

However, there does not seem any obvious way to easily extend these approaches to adding multiplication to the permissible operators in a similar manner. I am not sure if there exists a method to extend the approach to cope with additional operators for the Constraint Programming method. For both the recursion and Dynamic Programming approaches there are methods but they are considerably more complicated, since the operators are no longer associative with each other which makes the problem much harder. I now consider these methods.

4.1.2. Countdown Problem

Recursion

There are two different types of recursive methods: "bottom-up" algorithms, and "top-down" algorithms.

Bottom-up Algorithm

We pick an operator and two numbers from the set of numbers. We then evaluate the result obtained by combining the two numbers using the operator. The result is then placed back into the set of numbers, and the two numbers are removed from the set. If the result equals the target then we have found a solution, otherwise we recursively try to find a solution with the same target and the new set of numbers with n-1 elements. We repeat this for all possible choices of the operators and the pair of numbers.

Top-down Algorithm

We pick an operator and a partition of the six numbers into two subsets (e.g. the set [1, 6, 3, 9, 15, 20] might be partitioned into [1, 6, 9, 20] and [3, 15]). We recursively find all the targets attainable using one subset, and similarly all the targets attainable using the other subset. We then find *all* the values obtained by combining an attainable target from one subset and an attainable target from the other subset using the chosen operator. We then repeat this for all possible choices of the operator and all possible partitions of the original set of numbers.

Dynamic Programming

The dynamic programming approach for the Countdown problem is similar to the top-down algorithm, only that in the top down algorithm you have to re-compute the values at each level of recursion. For each subset of the set of numbers, it stores a list of all the targets attainable using the specified subset. For a subset of size 1, the list contains just that number. For a subset of size 2, the list contains the values you can get by combining the two numbers using one of the operators. In general, for a subset of size 'k', it looks at every possible way of splitting the set of numbers into 2 subsets. For each operator, and for each element from the list of targets attainable for one subset and for each element from the list of targets attainable from the other subset, it combines the two targets using the operator and stores the result in the list of targets attainable for the set of k numbers.

4.2 Choice of Algorithm

I have chosen to implement the recursive "bottom-up" algorithm. I chose that algorithm over the others for various reasons. Firstly, dynamic programming takes up a lot space as it stores the lists of targets attainable using any subset of the numbers. Since the number of cases considered is large, even for siz numbers, this could slow down the performance of the program quite considerably. Secondly, the top-down approach seems more complicated because you have to generate all the possible partitions. Also you have to store all the targets attainable for each of the subsets.

I will now describe my chosen algorithm in more detail.

Suppose the array s[] stores the n numbers which are being used. Each pair of numbers s_i and s_j is chosen in turn. Each of the operators is considered and s_i and s_j are combined using the operator. E.g. we evaluate $s^* = s_i + s_j$. The result s* is then tested to see if it is equal to the target. If it is, it is stored as a solution. If it is not, we remove s_i and s_j from the array and add s* to the array. We then recursively try to find a solution with the same target and the new array (which has only n-1 elements). We do this

for each pair of numbers and each operator. For reasons which will be explained in Section 4.4, we only have to consider each *unordered* pair of numbers, s_i and s_j, so we will only consider cases for which i<j.

In the algorithm below as in my program, instead of s_1, …, s_n, I will use nums[0] … nums[n-1].
The function combine searches for a possible solution using the array nums[0…n-1], and the given target. Ops is an array of the possible operators.

```
combine(n)
{
for (int i = 0; i < n-1; i++)              //choose i
  for (int j = i+1; j < n; j++)            //choose j
     for (int opNum = 0; opNum < ops.length; opNum++){    //choose operator
        int result = evaluate (ops[opNum], nums[i], nums[j]);

        if (result == target)    //another solution found
        {
           solution++;           //increase solution count and store solution
        }
     //save old values nums[i] and nums[j] – see section 4.3
     nums[i] = result;  //place result in the place of nums[i]
     nums[j] = nums[n-1];  //place the last element in the array in the place of nums[j]
     combine(n-1);  //call combine recursively with n-1
     //restore values nums[i] and nums[j] – see section 4.3
}
```

4.3 Optimisations

There were inefficiencies which I spotted after having testing my program. I noticed that sometimes the same solution was appearing more than once. I discovered I was separately counting solutions which were essentially the same, but in which some of the subexpressions were evaluated in a different order. The following example illustrates what is happening to the *nums* array at each level of recursion. Suppose the numbers are {3, 7, 1, 11, 20, 5} and the target is 72. The underlined numbers represent the result of an evaluation and bold numbers represent the numbers which are chosen.

When i = 0, and j = 2, and the operator is '+': [**3** 7 **1** 11 20 5]
When i = 1, and j = 3, and the operator is '+': [4 **7** 5 **11** 20]
When i = 0, and j = 1, and the operator is '*': [**4** **18** 5 20]
[72 20 5]

However, it is also possible to obtain the same elements in the array just by doing the operations in a different order.

When i = 1, and j = 3, and the operator is '+': [3 **7** 1 **11** 20 5]

When i = 1, and j = 3, and the operator is '+': [**3** 18 **1** 5 20]

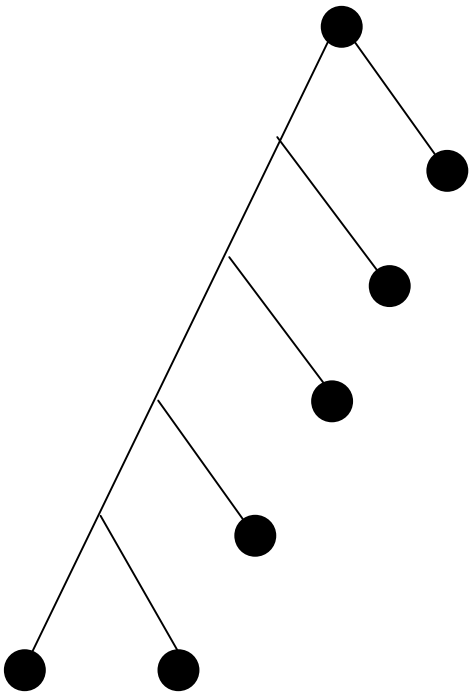
When i = 1, and j = 3, and the operator is '*': [**4** **18** 20 5]

[72 5 20]

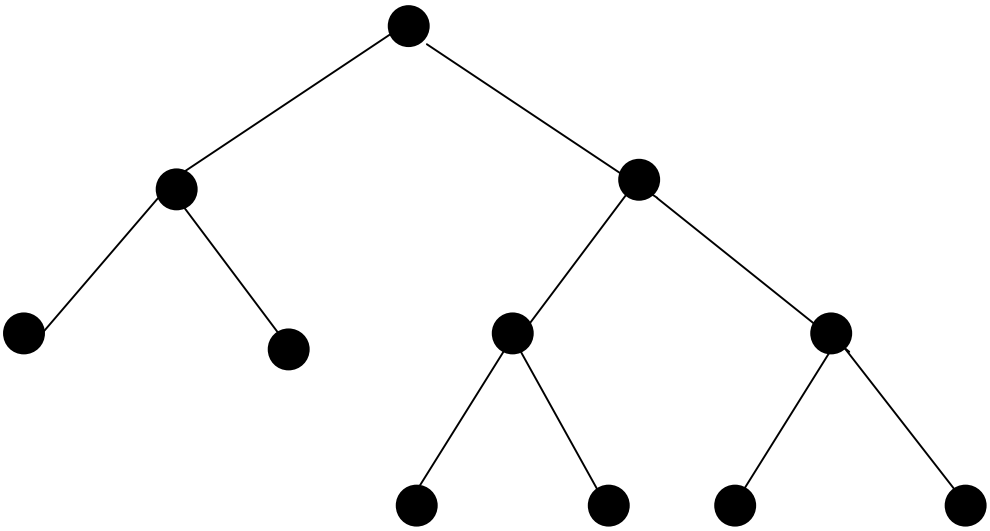
It is now clear that the same solution, (3+1) * (7+11), will arise in two different ways, producing solutions which are not really distinct from one another.

If we consider the expression tree corresponding to the expression (see Appendix B). In every expression tree there must be at least 2 leaf nodes (i.e. 2 of the original numbers) which gets combined directly with one another, as opposed to being combined with an intermediate result. There are two sorts of trees:

Tree 1

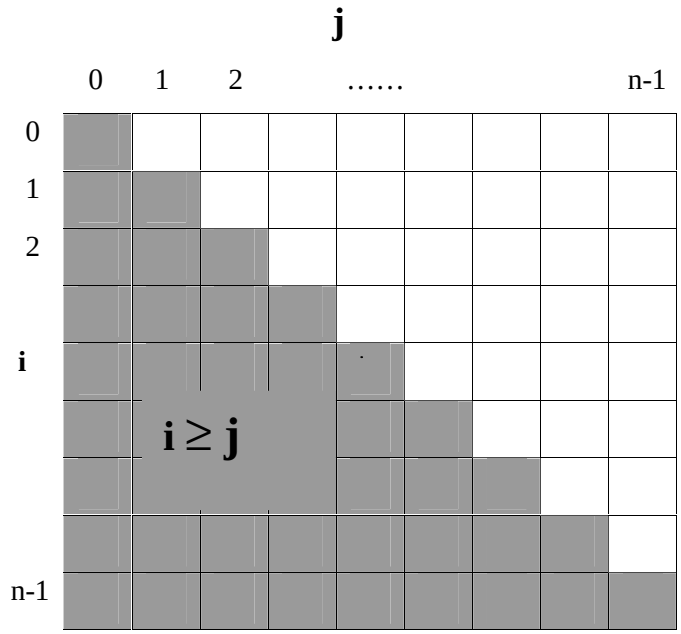


Tree 2



If the tree looks like tree 1, then the above problem will not affect it since there is only one pair of leaf nodes. However, if the tree looks like tree 2 (where there are 3 such pairs of leaf nodes), then it is possible for one pair of leaf nodes to be combined first and then get combined with another pair of leaf nodes which were combined later. It is therefore possible for the second pair of leaf nodes to be combined first and then be combined with the result of combining the other two leaf nodes. This of course produces equivalent expressions.

To solve the problem I envisaged all the pairs of i and j values as a matrix.



Each square represents a pair of numbers which could be combined. No pairs of i and j values in the shaded area could be combined as i ≥ j there, but we always have i less than j.

In order to avoid the above problem, I ensured that I always first combined the pair of leaves with the least value of j. It would then follow that when I combined two numbers, I could be sure that this pair of numbers had not previously been combined, in some call of combine(), at a higher level of recursion. This is true as we never look at a value of j that we have already considered a higher level of recursion – we always start from the value j+1. My optimised Countdown is called Countdown13, whereas the unoptimised version is called Countdown12.
It turned out more convenient to swap *i* and *j* loops around (i.e. nest the *i* loop inside the *j* loop) and to store the result of combining two numbers at the end of the array, removing the two numbers, rather than inserting it in the middle of the array – see Section 4.4 below.

4.4 Other Issues

When writing the code for my program, there were several decisions that had to be made on how I wanted the program to run. My decision would not be right or wrong, but sometimes I made the decision on the basis of what I thought would make the code most intelligible.

Backtracking

One major decision which I had to decide upon was how I was going to ensure that at each recursive call I had the correct values in the array. Since at each level of recursion the content of the array is different. I could copy the array and pass the new array of numbers as a parameter in the recursive call. Alternatively, I could use backtracking and save all the relevant values restoring the old values after the recursive call. I decided that I was going to use the backtracking method as I didn't think that it was efficient to have a different array at each level of recursion and copy it each time. In retrospect I am unsure whether it makes a difference with such a small value of n. If I had more time I would have investigated to see if there was in fact any significant difference between the two methods.

The backtracking method I used for Countdown13 differed slightly from that I used for Countdown12. For Countdown12, I stored the old values of nums[i] and nums[j] and then copied the result of evaluate() into nums[i] and moved the last element of the array, nums[n-1] into nums[j]. I then made the recursive call with n-1, i.e. nums[0…n-2].

```
combine(n, ){
    …
    int oldnumsi = nums[i]; //save value of nums[i] before recursive call
```

```
            int oldnumsj = nums[j]; //save value of nums[j] before recursive call
            nums[i] = result;
            nums[j] = nums[n-1];
            combine(n-1);
            nums[i] = oldnumsi;  //restore value of nums[i]
            nums[j] = oldnumsj;  //restore value of nums[j]
            …
        }
```

However, for Countdown13, it was easier to place the result at the end of the array, and negate the values of nums[i] and nums[j]. I would then ignore any negative values in the recursive call. We therefore needed the nums array to be of length 2n-1 to allow space for the result of each operation. Since one element has been added to the array, the recursive call will be combine(n+1, j+1). We pass j+1 as we only want to look at the next value of j with each value of i, as explained in Section 4.3.

```
    combine(n, jNext){
        …
        nums[i] = - nums[i];  //negate to stop reuse
        nums[j] = - nums[j];   //negate to stop reuse
        nums[n] = result; // place result at the end of array
        combine(n+1, j+1);
        nums[i] = - nums[i]; // un-negate for next iteration
        nums[j] = - nums[j]; // un-negate for next iteration
        …
    }
```

Legal Operations

In the Countdown numbers game you are only allowed to use integers, even as intermediary results. I therefore decided that if in the *evaluate* method a non-integer result was obtained (as might happen with division), I would return a –1 flag, which I would catch in the countdown method and ignore that value.

```
case '/':   //division
if (num1 >= num2)
  if (num1 % num2 == 0)
    return num1/num2;
```

```
  else return -1;
else
  if (num2 % num1 == 0 )
     return num2/num1;
  else return -1;
```

Similarly, negative intermediate results are not allowed, although this wouldn't make any difference to the targets attainable. Therefore, I always returned the absolute value of the difference between the two numbers. I also ensured that a zero difference could not be returned, and returned a –1 flag. This was in order to prevent division by zero.

```
case '-':
if (num1 > num2)
      return num1 – num2;
else if (num1 < num2)
      return num2 – num1;
else return –1;
```

This obviously means that if an invalid operation took place, e.g. 1/5, then this would ignore this selection of operator and operands and the recursion would not go deeper.

I also decided that I would stop the recursion when I found a solution (e.g. if it found a solution (3+4) *2, then it would stop, even though it would be possible that there was another solution ((3+4) *2) * (9-8)). I decided to do this as although there could be other solutions if the recursion unwound fully, I considered them trivial solutions, as they are essentially the same as solutions already found.

Although I thought that this was the most natural thing to do, a problem arose when I wanted to count how many cases the program might have to consider in the worst case. I was not counting all the possible cases, as there were cases which I was ignoring, because the recursion wasn't going to its maximum depth - as soon as I found a solution it would stop on that level of recursion. Some of the cases will also be ignored as a result of an 'illegal' operation returning a –1 flag (as described above). To get the true value of all the maximum number of cases that could be considered, I needed to unwind the recursion fully. The easiest way of doing this, with the least change to my program, was to make sure that the recursion did unwind in *every* situation. Although I could have decided to change my evaluate method, to ensure that all evaluations were legal, i.e. no –1 would be returned, I would then also have to make

sure that there was no division by zero, as this would cause an error. I therefore decided to specify the set of operators to be all plus operators i.e. 4 '+'s - there was therefore no worry of producing an invalid operation or a division by zero. I also had to make sure that a solution could never be reached, without using all the numbers, so I chose the numbers to all be 1's and the set the target to one more than the number of numbers used (i.e. the target was therefore unreachable with those numbers and those operators). In this way I was able to make a small change to my calling program in order to see how many cases were considered, for different values of n, the number of numbers in the set. This would provide a check on my analysis of the complexity of the algorithm, as I would be able to compare the number of cases considered with the predictions of my theoretical analysis.

Another decision that I had to make was to decide how many solutions I would search for. I could either stop searching as soon as I found one solution, or I could try and find all possible solutions with the given set of numbers. Although the actual Countdown game only requires players to find one solution, for my analysis of the algorithm and the game, I would need to see how many solutions there were altogether. However, as I was storing the solutions in an array, I would need to specify an upper limit to how many solutions I would store. I set the size of the array to 10,000, as in my test runs it was very unusual for six numbers to produce more than 10000 solutions. However, if the program was run with more than 6 numbers or the particular 6 numbers were ones which produced many solutions, then the program would only store the first 10,000 solutions, although the solution count would continue to count correctly even if there was over 10,000.

5. Program Design

5.1 General Issues

There were several key issues which had to be considered when designing the program.
Firstly, I had to decide what data structure I would use to store the numbers. There were a few different options available, such as a linked list or an array. I chose to store the numbers in an array, as although linked lists have several benefits over arrays, such as easier insertion and deletion, the backtracking to restore the list to its previous state could be quite tricky. Also, the process of doing the backtracking with linked lists would have taken more computation time and slowed the program down. In hindsight, it would not have been that much more complicated, and I would have needed fewer changes to my program to convert it to the optimised form. Given more time I would have investigated the differences in performance, if any, between using the two data structures.

A second design issue which had to be decided was the class structure I would use to implement the program. As it turned out, I developed two very similar Countdown programs – the optimised version and the un-optimised version. I therefore decided to have my two different versions of Countdown extend an abstract Countdown class which contains the parts of the constructor which are the same as well as the member variables and methods which are in common. I decided to implement it this way as I wanted to make my program as object oriented as possible, and minimise the amount of repeated code.

Thirdly I had to decide what the best way to store any solutions would be. In earlier versions of Countdown I stored the solutions as a list of the arithmetical computational steps which had been carried out to obtain the solution. For example, if the target was 518, and the numbers were [2, 3, 6, 10, 25, 100]. Then the target could be reached by the following steps.

Step 1: 25 * 2 = 50
Step 2: 50 * 10 = 500
Step 3: 3 * 6 = 18
Step 4: 500 + 18 = **518**

Although this seemed like a natural and clear way to display the results there were various problems with it. Firstly, printing out the solution as a series of steps took up a lot of space which wasn't so useful when printing to file. Also, I noticed that it was possible that there would be steps printed as part of the solution which weren't necessary for obtaining the target. For instance, if the target was 780, and the numbers were [2, 3, 7, 8, 10, 100], then one of the potential solutions was:

Step 1: 2 + 3 = 5
Step 2: 8 * 10 = 80

Step 3: 7 * 100 = 700

Step 4: 700 + 80 = 780

As shown above, it is possible for the solution to have steps which are not really used in the expression, e.g. 2 + 3 = 5.

Note: We would get exactly the same problem if we used a different operator, which would therefore produce a lot of "solutions" which in actual fact were not really different.

I therefore decided to store the expressions using expression trees, storing the numbers as expressions (see Appendix B). I created the expression trees using the following classes.

Constant – This is an expression which represents one of the original numbers.

BinaryExpression – This is an expression which is created by combining two expressions using one of the binary operators.

Expression – This is either a Constant or a BinaryExpression.

I have an array of expressions, expn, corresponding to the *nums* array. Each element in the *nums* array has a corresponding expression which shows how the number was obtained. Initially the array of expressions would contain six constants, i.e. the six chosen numbers. Every time two of the numbers are combined, e.g. nums[i] + nums[j], and put back into the array, the corresponding elements of the expression array are used to construct a new expression.

```
Expression expRes = new BinaryExpression(ops[opNum], expn[i], expn[j]);
```

This had various advantages. Firstly, the solution wouldn't contain sub-expressions which weren't really being used (like Step 1 in the example above). Secondly, the end result was an expression which was more meaningful then several statements which didn't necessarily lead on one from the next. It is also easy to print out the entire expression using a recursive function.

```
toString(){
    return "(" + left.toString() + op + right.toString() + ")";
}
```

It should be noted that the larger operand is printed out first in the expression, since for some operators, e.g. '-', I was evaluating the absolute value of the difference, and so I had to print it out in the correct way, i.e. 5 – 3, as opposed to 3 – 5.

Finally, I had to determine what data I would store to use for my statistical analysis of the problem. There are lots of different statistics which could be analysed from the Countdown problem, I therefore had to decide what data I would be collecting for my program and how I would use that data to produce a meaningful analysis of the problem. I also wanted to choose statistics about which I had a hypothesis to

see if the results actually obtained supported my hypothesis or not. I decided that there were two main areas for which I wanted to gather statistics.

Firstly, I wanted to analyse the Classical Countdown problem, to see if there was any difference in reaching the target depending on the number selection chosen (E.g. 4 Big 2 Small, 3 Big 3 Small, …, 6 Small). I therefore have to have some measure as to how 'successful' a number selection was. Obviously the number of solutions would be one way to compare different selections. I also chose to record the number of targets obtainable (i.e. given 6 numbers, how many of the targets from 1 – 1000 are obtainable). I represented this by an array of 1000 booleans, where the i^{th} element of the array was set to true if the program found a way of obtaining that number. With these statistics I would be able to compare different number selections and generate some results indicating the best strategy that one should employ in deciding how many big and small numbers to select. I also recorded the number of cases tried, and the computational time taken to find all solutions to see if there were any significant differences.

Secondly, I wanted to gather some statistics from the generalised version, to see if my theoretical analysis of the algorithm reflected the results obtained from tests using different sizes for the set of numbers. However, using more numbers would obviously increase the time it took to find all the solutions. This obviously constrained exactly how many numbers I could use and how many runs of each I could test.

5.2 Command Line version (CommandLine.java)

I tried to make the command line version of the program as general as possible, allowing the user as many options as possible, but not making it too complicated to use. I decided that I would give the user a choice of which version of Countdown to run, either Countdown 12 (un-optimised version) or Countdown 13 (optimised version). I then gave the user an option of either playing Classical Countdown i.e. the user would choose which number selection he/she wished (6 small, 5 small 1 big, 4 small 2 big, 3 small 3 big, 2 small 4 big), or the generalised Countdown game where the user could select how many numbers the game would be run with (e.g. 2 numbers or 7 numbers), and they could also decide if they want to choose those numbers randomly within a specific range or whether they want to choose specific numbers. They then may either choose the target randomly or choose a specific target.

After the search for the solution has finished, a list of statistics is printed to screen (as well as to the file results.txt). These include the number of solutions, the number of cases considered and the time taken.

The user is then given the option to print the solutions to file. If there was no solution, the nearest reachable values above and below the target is printed out to screen (as well as to file). The user is then asked if they want to play again or quit.
See Appendix C for example runs.

5.3 GUI version (gui3.java)

I wanted to provide as much functionality in the GUI as possible, but far too often a GUI can be crowded with too many buttons which over-complicate it and make it hard to use. The GUI automatically uses Countdown13, and runs using 6 numbers, although the user is given the option of either specifying the numbers or generating them randomly. If the user decides to generate them randomly, the user has to specify what the number selection is, e.g. 3 small 3 big, 4 small 2 big, etc, whereas, if the user wishes to specify the numbers, he/she may enter any numbers. The program then generates and displays a random target. The user may then start the search for solutions. If there are no solutions, the nearest solution above and below the target is displayed and the user is given the option of running the program again. If there are solutions, they are displayed one at a time is displayed at the bottom of the screen, and the user is asked if he/she wishes to see the next solution. Once the user has seen all solutions or doesn't wish to see any more solutions, they are given the option of running the program again.
See Appendix D for screen shots.

6. Algorithm Complexity

6.1 Complexity of the Problem

The subset sum problem is well known to be NP-Complete (Garey and Johnson). Although it seems likely that the Countdown problem is NP-Complete, unfortunately there does not seem any easy way to prove this. However, it is quite easy to show that, another special case, the subset product problem is NP-Complete.

The subset product problem (SPP) is defined as follows.

Instance: A positive integer T and n (not necessary distinct) positive integers whose sizes are s_1, s_2, …s_n.

Question: Is there a subset I of the set {1, 2,…, n} such that $\Pi_{i \in I} s_i = T$?

I am going to transform Exact Cover by 3-Sets (X3C) to the subset product problem. X3C is defined as follows (Garey and Johnson):

Instance: A finite set X with |X| = 3q and a collection C of 3-element subsets of X.

Question: Does C contain an *exact cover* for X, that is, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'?

Theorem SPP is NP-Complete.

SPP is obviously in NP, since we can guess the subset and check it in polynomial time.

The proof is a similar transformation to that used for the subset sum problem.

Let X be a set with |X| = 3q and C be a collection of 3-element subsets of X.

Denote $X = \{X_1, X_2, \ldots, X_{3q}\}$

Let $p_1, p_2, \ldots, p_{3q}$ be the first 3q prime numbers. Let the p_i element correspond to X_i where $1 \le i \le 3q$.

Let a given triple, $c \in C$ be $c = \{X_e, X_f, X_g\}$. Corresponding to c we define the number $s(c) = p_e * p_f * p_g$. The set $s_1, s_2, \ldots, s_n$ is the set of all such numbers s(c) one for each triple in c. The target T is defined by $T = \Pi_{1 \le i \le 3q} p_i$. It then follows that C has an exact cover for X, if and only if, there exists a subset of $s_1, s_2, \ldots, s_n$, such that the product of the numbers in the subset is T. This is true because we can select the number s(c) corresponding to c, and vice versa.

6.2 Complexity of the Algorithms

6.2.1. Un-optimised Version

The complexity of the algorithm is determined by the 3 nested loops in the function combine, which form the main part of the algorithm:

```
combine(n)
        for i=0 to n-2
          for j=i+1 to n-1
```

for op=1 to 4

combine(n-1)

Let T_n be the number of cases or expressions tried for a set of numbers. Then from the algorithm we obtain the following recurrence for T_n.

$T_n = \Sigma_{0 \le i \le n-2} \ \Sigma_{i \le j \le n-1} (4 * T_{n-1})$

For n = 1, there is only one case so $T_n = 1$. The recurrence then gives, $T_2 = 4$, as expected.

We now get

$T_n = 4\ T_{n-1} \Sigma_{0 \le i \le n-2} \ \Sigma_{i \le j \le n-1} (1) = 4\ T_{n-1} \Sigma_{0 \le i \le n-2} (n-i-1)$

$= 4\ T_{n-1} \Sigma_{1 \le p \le n-1} p,$ by putting p = n-i-1

$= 4\ T_{n-1} * \frac{1}{2}n(n-1)$

i.e, $T_n = 2n(n-1)T_{n-1}$

Expanding this

$T_n = [2n(n-1)]\ [2(n-1)(n-2)]\ldots.[2 \cdot 3 \cdot 2]\ [2 \cdot 2 \cdot 1]T_1$

So $\mathbf{T_n = 2^{n-1}\, n!\, (n-1)!}$

I can therefore work out the complexity T_n for different values of the number of numbers in the set (e.g., 2, 3, 4, 5, 6).

$T_2 = 2 * 2! * 1! = 4$

$T_3 = 4 * 3! * 2! = 48$

$T_4 = 8 * 4! * 3! = 1{,}152$

$T_5 = 16 * 5! * 4! = 46{,}080$

$T_6 = 32 * 6! * 5! = 2{,}764{,}800$

$T_7 = 64 * 7! * 6! = 232{,}243{,}200$

6.2.2. Optimised Version

Once again the complexity of the algorithm is determined by the 3 nested loops in the function combine, which form the main part of the algorithm. Combine now has two parameters and the first call is combine(n, 1).

combine(n, jNext){

```
        for  j = jNext to n
          for i = 0 to j
            for opNum = 1 to 4
            combine(n,j+1)
    }
```

Note: Although in the recursive call we pass j+1, we have in effect removed the i^{th} and j^{th} numbers, as we have negated them so that we skip over them. Therefore the column we wish to start at is j-1 (not j+1) as in our matrix (see Section 4.3) we have deleted two columns and added one new column. Therefore the column j+1 in the new matrix is column j-1.

From now on, I will use jx instead of jNext in this section. Let $T_{n, jx}$ be the number of cases considered in which no pair of numbers with I and j values less than jx are directly combined.

From the algorithm we obtain the following recurrence.

$$T_{n,jx} = 4 * \sum_{jx \le j \le n-1} \sum_{0 \le i \le j-1} T_{n-1,j-1}$$

Since $T_{n-1,j-1}$ doesn't depend on i:

$$T_{n,jx} = 4 \sum_{jx \le j \le n-1} T_{n-1,j-1} \sum_{0 \le i \le j-1} 1$$

$$T_{n,jx} = 4 \sum_{jx \le j \le n-1} j \cdot T_{n-1,j-1}$$

so, $T_{n,jx+1} = 4 \sum_{jx+1 \le j \le n-1} j \cdot T_{n-1,j-1}$

Subtracting these two equations we get

$\therefore T_{n,jx} = T_{n,jx+1} + 4\, jx\, T_{n-1,jx-1}$

If there are fewer than four operators used, say *c* operators, then the relevant recurrences in this and the previous section will be the same but have the 4's replaced by c.

To find the values of $T_{n, jx}$ we can envisage the following matrix (for n =6).

		jx						
		0	1	2	3	4	5	6
n	0	**0**						

1		**0**					
2			**0**				
3				**0**			
4		a			**0**		
5			**?**	.b		**0**	
6							**0**

Since $T_{n,jx}$ is the amount of work done, but only from starting from the jx^{th} column onwards, if there are *n* numbers, the last position in the array is n-1. So when j = n, there are no values as there is no column j, therefore the diagonal can be considered as zero's.

For $T_{n,0}$, since there is no pair of *i* and *j*, where *j* = 0, given that i < j. Therefore, starting from the 0^{th} column is the same as starting from the 1^{st} column (i.e. $T_{n,0} = T_{n,1}$). Writing *c* for the number of operators, we get

$T_{n,jx} = T_{n,jx+1} + c * jx\ T_{n-1,jx-1}$

The above diagram shows how $T_{n,\ jx}$ can be calculated from the two neighbouring values *a* and *b* where *a* is $T_{n-1,jx-1}$ and *b* is $T_{n,jx+1}$

Substituting in:

		jx						
		0	1	2	3	4	5	6
n	0	**0**						
	1	1	**0**					
	2	1c	1c	**0**				
	3	$3c^2$	$3c^2$	$2c^2$	**0**			
	4	$15c^3$	$15c^3$	$12c^3$	$6c^3$	**0**		
	5	$105c^4$	$105c^4$	$90c^4$	$60c^4$	$24c^4$	**0**	
	6	$945c^5$	$945c^5$	$840c^5$	$630c^5$	$360c^5$	$120c^5$	**0**

When running the Countdown program, I am interested in jx=1, and with four operators (i.e. c= 4). We therefore get the following results:

$T_{2,1} = 4$

$T_{3,1} = 48$

$T_{4,1} = 960$

$T_{5,1} = 26{,}880$

$T_{6,1}$ = 967,680

$T_{7,1}$ = 42,577,920 (not shown in the table)

I checked the values that I got from the number of cases considered, for both Countdown12 and Countdown13, when allowing the recursion to fully unwind to see if they were the same as these values. Checking the results obtained by my program, see Appendix C, they are exactly the same as the values calculated here. This is encouraging as it means that the values obtained from my program are consistent with the theoretical analysis of the algorithm. It should be noted that the value of cases considered for 4 numbers is given by "cases considered for 4 numbers", as opposed to "Number of cases considered". This is because the complexity analysis calculated for each number, is how many cases have to be considered just using that number of numbers, e.g. 6. However, the "Number of cases considered" is the total number of cases considered using 2 numbers, 3 numbers, …, 6 numbers.

T_n and $T_{n,1}$ will be upper bounds on the number of cases considered for Countdown12 and Countdown13 because in practice the recursion is stopped early, either because of an illegal operation, or because the target is found without using all the numbers.

It is clear from the results above that the bounds for Countdown13 is significantly less than that for Countdown12. Moreover, the ratio of the bounds increases with n, as shown below.

n	2	3	4	5	6	7
$T_n/T_{n,1}$	1	1	1.2	1.7	2.9	5.5

7. Testing

7.1 Analysis of Sample Data

I will use *n* to denote the number of numbers in the set. There are many statistics that I could have gathered from my program, however due to lack of time I have not managed to do all of them. However, I have gathered some statistics which I feel are relevant in analysing the performance of Countdown13 (optimised version) against Countdown12 (un-optimised version). I have also collected data to find out how many times out of 100 runs a target was reachable. This is a good indicator to compare the different number selections (i.e. 2 small 4 big, 3 small 3 big, etc.) to see whether one is more likely to reach the target than the others.

I initially intended to do 1000 runs for each statistic and then average the results, which wasn't too bad for Countdown13. However, for Countdown12 it was taking too long (1.5 hours) to compute. I therefore only did 100 runs for each case.

I collected the same statistics for Countdown12 and Countdown13, apart from the Targets Achieved which I only did for Countdown13.

There were 5 statistics which I computed for my Classical Countdown program. All these results are included in Appendix E.

- Targets Achieved for the different number selections

I analysed the number of targets that were reached depending on how many of the six numbers were used. I then plotted a graph of the number of targets that were achieved against how many numbers. For instance, with 3 small 3 big number selection,

Using 2 numbers: 0 targets achieved

Using 3 numbers: 6 targets achieved

This means that just using 2 numbers out of the six, it wasn't ever possible to reach the target. Whereas using 3 numbers out of the six, it was possible to reach the target 6 times out of 100. I did this for each different number selection.

- Average number of solutions for the different numbers selections

I analysed the number of solutions that were found using each of the different number selections.

E.g. 3 small 3 big – 456 solutions found

4 small 2 big – 512 solutions found

Note that, as mentioned in Section 3, the number of solutions is not such a good indicator as this may include repetitions.

- Average running time for the different numbers selections

  I analysed the running time for each of the different number selections

- Average number of cases considered for the different numbers selections

  I analysed the number of cases considered for each of the different number selections. I assumed that this wouldn't change much, as the only factor which causes the number of cases considered to vary, is if the recursion breaks early, due to an illegal operation, or a target obtained using less than 6 numbers. Therefore, I suspected that there wouldn't be much variation depending on the different number selections.

- Average number of targets obtainable for the different numbers selections

I analysed the number of targets obtainable, in the range of 1-1000, for each of the different number selections.

E.g. 3 small 3 big – 890 targets obtainable

4 small 2 big – 920 targets obtainable

This means that when using 3 small and 3 big numbers, 890 targets out of the possible 1000 were attainable with those numbers.

There were 4 statistics which I calculated for my generalised countdown program, for different values of n (between 2-7).

- Average number of solutions for each n

  I analysed the number of solutions for different amount of numbers used.

- Average running time for each n

  I analysed the running time for different amount of numbers used.

  I was hoping that there would be a strong relationship between the running time and the number of cases considered. As you would assume that one would depend on the other.

- Average number of cases considered for each n

I analysed the number of cases considered for different amount of number used.

- Average number of targets obtainable for each n

  I analysed the number of targets obtainable for different amount of numbers used. This was a good indication of the range of targets that was generally attainable using that number of numbers.

  Just to highlight the difference between Targets Achieved and Targets Obtainable. Targets achieved specifies in how many of the 100 runs the target was reached. Targets obtainable means, how many targets between 1 and 1000, can be reached using the specified *n* numbers, averaged over 100 runs.

7.2 Conclusions from sample data

I used Microsoft Excel to plot graphs illustrating some of the results. The graphs can be found in Appendix E.

There were several conclusions that I arrived at from studying the statistics produced by my sample data.

Firstly, I anticipated from my complexity analysis of the algorithm that, for n=6, the optimised version would run about 3 times as fast as the un-optimised version. I was able to check this with the runs I did for both versions and compare the running time and the number of cases considered. From looking at the graphs which compare the cases considered and the running times of Countdown12 and Countdown13, it is clear that Countdown13 runs roughly 3 times as fast, for n=6. I also saw that as *n* increases (i.e. from 6 to 7) the ratio of the number of cases considered increases. Thus Countdown13 does about 1/3 of the amount of work that Countdown12 does (for n=6) and this fraction decreases as *n* increases.

The table below shows the ratio between Countdown12 and Countdown13 for cases considered (CC) and running time (RT), for different values of *n*.

n	2	3	4	5	6
CC12/CC13	1	1	1.2	1.7	2.8
RT12/RT13	1	1	1	1.7	3.1

I next looked at the statistics relating to the different number selections (e.g. 3 big 3 small, 4 big 2 small) for both Countdown12 and Countdown13, using the Classical Countdown.

I concluded that the number of solutions didn't vary that much depending on the number selection. The same was true for the cases considered, running time and targets obtainable. Although this was surprising at first, after some thought it seemed reasonable. The running time is obviously dependent on the number of cases considered. The number of cases considered will vary depending on how many "illegal operations" and how many times it finds a solution without using all six numbers. These probably won't vary much whether 6 small numbers are used rather than 2 small and 4 big numbers. However, what was surprising was that there wasn't more variation in the number of targets obtainable (i.e. how many targets, from 1 –1000, can be obtained with the 6 numbers). I had thought that the number of targets obtainable would be greater if there was a mixture of big and small numbers, as if all the numbers were small then it might be hard to obtain the larger targets. Similarly, it would be hard to achieve the smaller targets if there were too many big numbers. Obviously, my hypothesis was false and the number selection chosen doesn't make such a difference.

I then looked at the statistics for targets achieved, depending on how many of the 6 numbers were used for the different number selections. There did not seem to be so much variation between the different number selections. All of which gave roughly the same percentage as with the following table.

Numbers used	2	3	4	5	6
Targets Achieved	0%	7%	45%	70%	85%

However, although in general there was not a radical variation between the results, I did notice that at both extremes of the number selections (i.e. 6 small, 2 small and 4 big) fewer targets were achieved for each value of 'numbers used'. The central selection of 4 small 2 big seemed to produce the most targets achieved. As explained earlier, this is what one would expect, as a balance of small and big numbers would seem to have the best chance of reaching the target.

I then looked at the statistics for the runs of the generalised version. I obviously did expect that there would be a significant variation between cases considered, running time, and number of solutions for

different values of *n*. Unfortunately, I wasn't able to run Countdown12 for more than 6 numbers, as it took too long.

As mentioned before, the number of cases considered includes those using fewer numbers. This accounts for the apparent anomaly, for n = 3 using Countdown13 where the number of cases considered is 52, but $T_{3,1}$ is 48.

As you can see from the graph comparing Countdown13 for 2 to 6 numbers, there is a huge jump from n = 5 to n = 6 numbers. For 5 numbers there are 21407 cases considered for Countdown13, whereas there are 658924 cases considered for 6 numbers – 30 times as many. However, the graph for Countdown13 for 2 to 7 numbers shows, the number of cases considered is 26028025, which is 40 times that for 6 numbers. Correspondingly the running time for 7 numbers is 57085 milliseconds, as opposed to 1508 milliseconds for 6 numbers – almost 40 times as long.

We can see that for 7 numbers the number of targets which are obtainable is 981 out of 1000. This implies that given 7 random numbers and a target, it is almost certainly possible to reach the target. The 7 numbers were chosen in the range 1 – 10. However, it is of course possible that if the range was a lot larger, e.g. 1 – 100, it might be difficult to reach small targets.

I think that my results show that when producing the Countdown numbers game, they designed it perfectly. If 7 numbers were allowed, it would make the game too easy, and if only 5 numbers were allowed then the game might be quite difficult. Also, the choice between small numbers and big numbers is appropriate, as the big numbers allow you to reach the neighbourhood of high targets, whilst the small numbers allow for fine tuning to reach the target exactly.

8. Results and Conclusion

8.1 Does the project meet the objectives?

I had five objectives as outlined in Chapter 3. I believe that my project met 4 out of the five (although I did state that I would like to achieve the fifth objective *if time was available*).
The first objective was met as I did write an algorithm which managed to find an exact solution of the numbers game.

The second objective was achieved as I managed to analyse the complexity of both versions (Countdown12 and Countdown13) of the numbers game, I couldn't find a closed form for the complexity of the optimised version. Although I wasn't able to prove that the Countdown problem in its general form is NP-complete, I managed to show that both the subset sum and subset product are both NP-complete.

The third objective was met as I did provide a user-friendly GUI to run the program.

The fourth objective was accomplished as I ran both versions of the program with sample data, and analysed those results, and I came up with some conclusions as to what those results mean.

As mentioned earlier, I didn't have enough time to try and achieve the fifth objective, which was to try and find some heuristic methods to solve the Countdown problem, when the set of numbers was too large for an exhaustive search to be used. For example, one possible method could be along the following lines; suppose the target is 649 and there is one big number which is 100. One would try to find a multiplier of the large number, which is nearest to the target. Since 649 is in between 600 and 700, one would try and express the target in the form of either

$$649 = (100 + x) * 6 \pm y$$

or

$$649 = (100 - x) * 7 \pm y$$

Overall my project does solve the Classical Countdown problem (i.e. with 6 numbers). Although I am sure my program could be optimised further, my optimised version (Countdown13) reduces the work by a factor of 3 compared to the un-optimised version.

8.2 Further Work

Unfortunately there are several additional things that I would have liked to investigate and analyse, but due to time constraints I have been unable to. Below are listed a few things that I would have liked to investigate given the time.

Firstly, I noticed that my program was still producing repeated solutions, when the solution did not use all of the numbers. It would be interesting to see if any more optimisations were possible to reduce all repeated solutions, so that the solutions obtained were in actual fact all distinct solutions. For instance, to try and deal with the associativity of addition and multiplication, to stop repeated solutions such as (3 + 4) + 5), as well as (3 + (4 + 5) which are essentially the same solution.

Secondly, testing more than 7 numbers took too long with my program. One possible way to get round this would be to this would be to stop the search when only one solution was found, this way it would be possible to run countdown with many more numbers, provided enough solutions existed.

Thirdly, it would be interesting to compare different algorithms (such as top-down recursive version, dynamic programming, constraint programming, etc.) for solving both the classical countdown and generalized problem, and see how they fare in terms of time, number of cases considered, and to investigate possible optimisations of the algorithms.

Fourthly, when there is no exact solution, it would be interesting to see how close on average the nearest solution above and below is to the target.

Fifthly, I only had time to run the program with all the standard four operators (apart from when I ran it with all '+' for checking my theoretical analysis of the algorithm. However, it would be interesting to see what would happen if only a subset of the four operators were allowed, e.g. only multiplication and addition, and to see if that would have a significant effect on the statistics already obtained. From the results of such experiments it might be possible to investigate what are the operators most commonly used within a solution.

Finally, as mentioned in my objectives, as the Countdown problem seems to be NP-Complete, although I haven't managed to prove it, it would be interesting to investigate possible heuristic methods of finding a solution in the general case when exhaustive search methods take too long.

References

Baase Sarah, Computer Algorithms: Introduction to design and analysis, Reading, MA: Addison Wesley Longman, 2000.

Bishop Judith, Java Gently, Pearson Education, Third Edition, 2001.

Garey & Johnson, Computers and Intractability: A guide to the theory of NP-Completeness, W.H.Freeman and Company, New York, 1979.

Prohl Ian & McDowell Charlie, Java by dissection: The essentials of Java programming, Addison-Wesley Pub Co, 1999.

Web Sites

Eck David, Introduction to Programming Using Java, On-line textbook.
http://math.hws.edu/javanotes

http://www.channel4.com/entertainment/countdown_game/NumbersGame.html

http://www.neil.parkwaycc.co.uk/numbers

http://www.brothercake.com/games/countdown.shtml